A3: Automatic Asynchronous Aggregation

ABSTRACT

Synchronous aggregation in parallel and distributed computing is the root reason of low efficiency. On the other hand, asynchronous aggregation is promising, which breaks up synchronization barriers and allows coordination-free execution. However, asynchronous aggregation suffers from the problems of non-guaranteed correctness and non-deterministic performance due to the possible redundant computations and communications. Moreover, asynchronous program is a lot harder to write, tune, and debug. To address these issues, we propose A3, Automatic Asynchronous Aggregation. Based on A3, we design and implement a Datalog system, A3Log, which provides both shared-memory runtime engine and distributed runtime engine. Our results on Amazon EC2 show that asynchronous execution can achieve up to two-orders-of-magnitude speedup over synchronous counterpart.

KEYWORDS

Asynchronous aggregation, parallel aggregation, recursive program, Datalog

ACM Reference Format:
. 2017. A3: Automatic Asynchronous Aggregation. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In general, a sequential computation can be expressed as an ordered list of operations on input data. Among these operations, *aggregate operation*, such as MAX/MIN/SUM, has for purpose the simultaneous use of different pieces of information in order to come to a conclusion or a decision, which is a fundamental operation in data analytics, data mining, machine learning, decision making, etc.

Aggregation implies a dependency relationship between data items and coordination between concurrently executing computations. The aggregate (or group-by aggregate) computation depends on its inputs which can be the outputs of previous computations, so it has to coordinate with other computations and waits for *all* its inputs to be ready. Therefore, a *synchronization* step is required to guarantee the correctness of aggregation. On the contrary, as long as NO aggregation exists in the list of operations, the computations can be embarrassingly parallel.

Synchronization is the unavoidable coordination step for aggregation though it is known to be costly [8]. Researchers have made great efforts in improving parallelism while reducing the impact of synchronous aggregation. But the costly synchronization for aggregation is required to obtain a correct result. The parallelism can only be achieved between synchronization barriers but not across synchronization barriers, which reduces the scope of parallelism. For example, MapReduce [18] decomposes a computation into (non-aggregate) map operations and (aggregate) reduce operations so as to parallelize the non-aggregate map operations and parallelize the reduce operations separately. But the synchronization barrier before reduce phase is indispensable due to the aggregation property. Synchronization does not only reduce resource utilization and parallelism but also sacrifices potential scalability. In parallel and distributed computing, synchronization is the key to guarantee correctness (or the consistency of the ordered operations) while at the same time is the key to limit efficiency. Therefore, *synchronous aggregation is the root reason of low efficiency.*

On the other hand, *asynchronous aggregation* is a promising technique in which the aggregation is performed on part of inputs instead of the complete set of inputs, such that the next computation can continue based on the partial aggregation results. The strict order of operations on data including aggregations is not required. In other words, asynchronous aggregation ignores synchronization barriers and allows coordination-free execution, which is a nice property in parallel and distributed computing.

Then, a few questions related to 1) correctness, 2) efficiency, and 3) automatability could be posed. Can asynchronous aggregation return the exactly *same* result as synchronous aggregation? Can asynchronous aggregation be guaranteed to achieve *better performance* than synchronous aggregation? Can a program with synchronous aggregation be *automatically* converted to that with asynchronous aggregation? Our answers are triple YES, or at least under some clearly defined conditions. Given the positive answers, we first analyze the pros and cons of asynchronous aggregation:

Pro 1: Coordination-Free Execution. In parallel and distributed computing, synchronous aggregation needs coordination between processers to ensure that all its inputs are ready, where the inputs may come from other processers. Asynchronous aggregation is allowed to be performed on part of inputs, so it is unnecessary to block and wait. Every processor independently executes operations based on its available inputs, where no coordination or synchronization is required across processors. By minimizing coordination between concurrently running processers, the scalability and performance can be maximized.

Conference'17, July 2017, Washington, DC, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Pro 2: Early Result Return. Synchronous aggregation implies that the next operation will not start until all inputs have been completely aggregated. If the aggregation is stuck due to heavy load, the following concatenation operations will be delayed. Asynchronous aggregation allows next operation to early start and to use the most recent information (partial aggregation result) to proceed, so it is expected to return an approximate result earlier. Online aggregation [25] is such an example.

Pro 3: More Effectiveness of Aggregations. The effectiveness of aggregate operations in asynchronous aggregation is an important advantage but tends to be underestimated. Synchronous aggregation requires to aggregate the information from previous iteration, while asynchronous aggregation allows to use the most up-to-date information. Thus, the aggregate operation is expected to be more effective.

Pro 4: Scheduling Flexibility. Prior works [9, 54] have pointed out that not all inputs contribute equally to the outputs. By prioritized scheduling of the computations that most affect its output, the result can be returned much faster or much more accurate. Synchronous aggregation limits the scheduling due to the synchronization barriers, while asynchronous aggregation will introduce more scheduling flexibility so that even higher efficiency can be explored.

Con 1: Non-Guaranteed Correctness. A number of previous works have applied asynchronous aggregation to accelerate some iterative computations while obtaining correct results, including GraphLab [33], Myria [49], Giraph++ [46], GiraphUC [23], GRACE [48], and GunRock [50]. But of course, not all computations are suitable for asynchronous aggregation. Some may lead to wrong results. These previous works did not clearly identify the conditions of returning the correct results. Maiter [55] provides the conditions for correct asynchronous graph computations, but these conditions are not general enough to judge arbitrary aggregate operations. If we blindly use asynchronous aggregation for all computations, the correctness cannot be guaranteed.

Con 2: Non-Deterministic Performance. Asynchronous aggregations are free from synchronizations. This implies that the computations and communications are not under control any more, which may lead to redundant computations and communications and potentially reduces the efficiency. As mentioned in Pro 3, asynchronous aggregation gives us scheduling flexibility, but at the same time, it leaves us with non-deterministic performance.

Con 3: More Complexity. Programmers are used to write sequential programs or synchronous parallel programs. Had the asynchronization conditions been formally given, it is still hard for a non-expert programmer to manually verify these conditions from their programs. In addition, writing asynchronous programs and designing asynchronous systems are even harder, because asynchronous implies disorganized and as a result complicated. Experience from Google [12] strongly suggests a synchronous programming model, since asynchronous code is a lot harder to write, tune, and debug.

In order to benefit from the pros while minimizing the cons, we propose *A3*, *Automatic Asynchronous Aggregation*. A3 solves the problem in the context of a very common class of computations, *recursive computations*. Recursive computation repeatedly applies the same operations on the output of previous recursion, where the repeatedly applied operations may include aggregation. Recursive computations are commonly used in graph algorithms (e.g., PageRank), machine learning (e.g., Belief Propagation), and science computing (e.g., Jacobi Method). In summary, this paper offers the following contributions:

Contribution 1: A3 Theory. A3 theory addresses the above three cons. 1) To guarantee the correctness, we clearly define the conditions that guarantee asynchronous aggregation to return the same result as synchronous aggregation. Even for some recursive programs that do not satisfy these conditions, we propose an approach to conditionally convert them to be qualified for asynchronous aggregation. 2) To guarantee the better performance, we propose a scheduling policy, which is theoretically proved to return result faster. 3) To alleviate the burden of programmers, we propose an automatic asynchronization technique. User's sequential program can be automatically checked for asynchronization possibilities and can even be automatically converted to the asynchronous program. (Sec. 3)

Contribution 2: A3Log System. We propose a Datalog system, *A3Log*, to support automatic asynchronous aggregation. Datalog is an excellent candidate language for expressing computation logic because of its high-level *declarative* semantics, exposing plenty of opportunities for automatic parallelization and automatic asynchronization. The A3 theory results are embedded in the Condition Checker component, so that it can asynchronize user's program automatically. A3Log provides both shared-memory runtime engine and distributed runtime engine. (Sec. 4)

Contribution 3: Extensive Experimental Studies. We experimentally evaluate A3Log, compared with 1) Socialite [30, 39], an open-source Datalog system that supports monotonic recursive aggregation, 2) GraphLab [33], a graph system that supports asynchronous computation, 3) Myria [49], an open-source Datalog system that supports asynchronous execution, 4) Maiter [55], a graph based system that supports asynchronous computation. We also perform evaluations with 7 algorithms and on 20 more datasets. The experiments are performed on a 32-core instance for many-core experiments and on a cluster with 64 instances for distributed experiments. Our results show that A3Log outperforms other systems in many-core experiments and shows comparable performance with Maiter in distributed experiments. Our results also show that the asynchronous execution of A3Log can achieve 2.25X-222.82X speedup over the synchronous version for various datasets. (Sec. 5)

Besides, we analyze the aggregate and non-aggregate operations in Sec. 2. We review the related works in Sec. 6. We then conclude the paper in Sec. 7.

2 REVISIT AGGREGATE OPERATION

A *sequential* computation (as opposed to parallel/distributed computation) is composed of a sequence of operations on input. These operations can be classified into two categories, the aggregate operations and the non-aggregate operations. We focus on numerical aggregation in this paper.

2.1 Aggregate/Non-Aggregate Operations

An ***aggregate operation*** is a function $g()$ that requires more than one variables as input, i.e., $g(X)$ where $X = \{x_1, x_2, \ldots, x_n\}$, and outputs zero or more values. For example, MAX, MIN, SUM, AVG, and COUNT are commonly used aggregate operations. A special but common case is the ***group-by aggregation***. A set of input key-value pairs are grouped by key, and the values in each group are aggregated to obtain a new set of key-value pairs. In other words, the group-by aggregation is composed of multiple subset aggregate operations.

On the contrary, a ***non-aggregate operation*** is a function that takes only one value as input, i.e., $f(x)$, and outputs zero or more values. Since non-aggregate operation only takes one value unit as input, it has the *distributive* property, i.e., $f(x \cup y) = f(x) \cup f(y)$. In essence, the aggregate operation implies a *data fusion*, while a non-aggregate operation implies a *data transformation*.

Note that, the aggregate and non-aggregate operation should be judged according to the number of *variable* inputs instead of constant inputs. For example, the join operation in relational algebra is in general an aggregation since it merges two data sets into one. It takes two inputs A and B and produces one output. However in distributed join implementation, the whole data set B can be small enough to be cached on all distributed workers, which implies that B is a constant input for the distributed join operations. By partitioning and distributing A, the join operation is achieved by merging a part of A and the locally cached whole B on each worker. In such a case, the join operation is considered as a non-aggregate operation because each distributed join operation only takes a part of A as a variable input which is different on different processor, where B is the constant for all processors.

Parallelization. Non-aggregate operation only takes one input and does not rely on any other variable, which makes it have the distributive property. By partitioning and distributing the input to many workers, the non-aggregate operations can be embarrassingly parallel *without communication* between workers. Aggregate operation takes multiple inputs which may originate from other workers, so that these aggregate operations or the group-by aggregation can be parallelized but *with communication* between workers. If the inputs for multiple aggregate operations are disjoint, these aggregate operations can be parallelized without communication by carefully partitioning the inputs. However, this is usually impossible since the inputs probably result from the previous computations on other workers and cannot be particularly partitioned.

2.2 Recursive Aggregation

Recursive aggregation is a sequence of interleaving aggregate and non-aggregate operations where all the aggregate operations are the same and all the non-aggregate operations are the same[1]. It is very common in graph algorithms, data mining and machine learning algorithms. The program with recursive aggregation is referred to as *recursive program*. In this paper, we will focus on optimizing the computations with recursive aggregation.

Let X denote a set of input variables of aggregate operation and x denote the input of non-aggregate operation. A recursive program can be represented as follows.

$$\begin{aligned} X^{k+1} &= f(x^k), \\ x^{k+1} &= g(X^{k+1}), \end{aligned} \tag{1}$$

It starts from x^0, and x^k is the result after k recursions. It can also start from X^0 in which case the update order is exchanged and an aggregation is first applied. This recursive program terminates when there is no difference between X^{k+1} and X^k or the difference between x^{k+1} and x^k is small enough. Let $(g \circ f)^n$ denote n applications of $(g \circ f)$. The final result of the recursive program is $(g \circ f)^n(x^0)$. Suppose x contains a set of data items, the $f()$ operation can be parallelized with each working on a subset. The aggregate operation $g()$ can be parallelized if it is a group-by aggregation, but these parallel aggregate operations are synchronized to wait for the complete set of X^{k+1}.

Example 1: SSSP. The single source shortest path (SSSP) computation is a recursive program that derives the shortest distance from a source node to all other nodes in a graph. The shortest distance is initialized as $+\infty$ for each node except for the source, which is initialized as 0. The $f()$ operation for a node i takes a tuple $\langle i, d_i^k \rangle$ as input where d_i^k is the shortest distance in the kth recursion, computes $f(d_i^k) = d_i^k + w_{i,j} = td_j^{k+1}$ for any outgoing neighbors j (where $w_{i,j}$ is the distance from node i to j), and outputs the tuples set $\{\langle j, td_j^{k+1} \rangle\}$ and its own $\langle i, d_i^k \rangle$. The aggregate operation $g()$ with respect to each node j takes the input tuples $\{\langle j, td_j^{k+1} \rangle\}$, performs aggregation $g(td_j^{k+1}) = min(\{td_j^{k+1}\}, d_j^k) = d_j^{k+1}$, and outputs $\langle j, d_j^{k+1} \rangle$. It terminates when all nodes' shortest distances are not changed from previous recursion.

Example 2: PageRank. The PageRank computation is another typical recursive program for ranking the nodes in a graph. The ranking score is initialized as $r_i^0 = 1/|V|$ for each node i where $|V|$ is the total number of nodes. The $f()$ operation for node i takes a tuple $\langle i, r_i^k \rangle$ as input where r_i^k is the ranking score in the kth recursion, computes $f(r_i^k) = 0.85 * r_i^k / d_i = tr_j^{k+1}$ for any outgoing neighbors j (where 0.85 is the constant damping factor and d_i is the out-degree of node i), and outputs the tuples set $\{\langle j, tr_j^{k+1} \rangle\}$. The aggregate operation $g()$ with respect to each node j takes the input tuples $\{\langle j, tr_j^{k+1} \rangle\}$, performs aggregation

[1]There are also recursive programs without aggregations, which can be embarrassingly parallel.

$g(\{tr_j^{k+1}\}) = \sum_j tr_j^{k+1} + 0.15 = r_j^{k+1}$ and outputs $\langle j, r_j^{k+1} \rangle$. It terminates when the difference between two continuous recursions' ranking scores is small enough.

3 A3 THEORY

In this section, we first formally define asynchronous aggregation and introduce accumulated recursive program. We then present how A3 addresses the three concerns of asynchronous aggregation.

3.1 Asynchronous Aggregation

The recursive program in Equation (1) is defined with synchronous aggregation. In parallel or distributed computing, the *synchronous* aggregation means that all the parallel $f()$ operations have to be completed before the next $g()$ operation starts. The $g()$ operation has to be applied on the whole set X^k. Correspondingly, we define asynchronous aggregation as follows.

Definition 3.1. (**Asynchronous Aggregation**) In a recursive program, we assume that the set X^k from any kth recursion is partitioned into m disjoint subsets, i.e., $X^k = \{X_1^k, X_2^k, \ldots, X_m^k\}$, and $\forall i, j, X_i^k \cap X_j^k = \emptyset$. Asynchronous aggregation is to aggregate multiple subsets from different recursions, i.e., $g(X_i^k \cup X_j^l)$ where $k \neq l$.

From the definition of recursive aggregation as shown in Equation (1), X^{k+1} is resulted from x^k, and x^k is resulted from X^k. X^{k+1} does not exist before applying $g()$ on the complete set of X^k. Thus, the asynchronous aggregation $g(X_i^k \cup X_j^{k+1})$ is only possible after X^{k+1} is obtained. In other words, X^{k+1} is a *replacement* of X^k and is closer to the final result. The asynchronous aggregation of X^{k+1} and X^k may result in a wrong result since X^k is a replaced result which is not supposed to be aggregated.

Semi-asynchronous. The asynchronous aggregation is possible for the group-by aggregation which contains multiple sub-aggregations, each associated with a group. Suppose a set of sub-aggregation results x_i^k only depend on a subset of X^k, i.e., $x_i^k = g(X_i^k)$ where X_i^k is a subset of X^k. As long as the subset X_i^k (but not necessarily the whole X^k) is available, the aggregation $g_k(X_i^k)$ can be performed to obtain x_i^k without waiting for the other subsets X_j^k $(j \neq i)$. However strictly speaking, this is not asynchronous aggregation because it is not an aggregation over multiple recursions' results. The computation can proceed with x_i^k and further obtain X_i^{k+1} after applying $f()$ operation. Similarly, X_i^{k+2} can be obtained as long as their dependencies are available. We refer to this sort of dependency-based asynchronous aggregation as *semi-asynchronous* aggregation. Semi-asynchronous aggregation can guarantee the correctness and is shown better performance in some use cases [33, 44]. However, the asynchrony is still limited by the dependency scope.

3.2 Accumulated Recursive Program

In order to make asynchronous aggregation feasible, we introduce a particular type of recursive programs.

Definition 3.2. (**Accumulated Recursive Program**) Accumulated recursive program is represented as follows.

$$\begin{aligned} \Delta X^{k+1} &= f(\Delta x^k), \\ \Delta x^{k+1} &= g(\Delta X^{k+1}). \end{aligned} \tag{2}$$

It starts from $\Delta x^0 = x^0$ and terminates when Δx^k is 0 or small enough. The final result is the aggregation of all intermediate results, i.e., $x^{k+1} = g(\Delta x^0, \Delta x^1, \ldots, \Delta x^{k+1})$. Alternatively, the result can be represented as follows.

$$g\Big(\Delta X^0 \cup (f \circ g)(\Delta X^0) \cup \ldots \cup (f \circ g)^k(\Delta X^0)\Big). \tag{3}$$

Intuitively, the accumulated recursive program performs the same computation as normal recursive program, but the final result is the aggregation of all the intermediate results as shown in Equation (3). The asynchronous aggregation becomes meaningful in accumulated recursive programs, since the final result is the *aggregation* of all recursions' results but not the last recursion's result. Note that, ΔX^k represents the *increment* from previous recursion. This implies that the accumulated recursive program exhibits **monotonicity**, which has been well studied in the literature [3, 24, 30, 49]. Briefly speaking, monotonicity means that we only add information, never negate or take away. However, not all recursive programs are monotonic. We have the following theorem to guide the transformation.

THEOREM 3.3. *(**Monotonizability**) The accumulated recursive program defined in Equation (2) will return the same result as the normal recursive program defined in Equation (1), as long as the the following conditions are satisfied.*

- ***monotonic:*** *$X \subseteq f \circ g(X)$;*
- ***accumulative:*** *$g(X_1 \cup X_2) = g(g(X_1) \cup X_2)$;*
- ***commutative:*** *$g(X_1 \cup X_2) = g(X_2 \cup X_1)$;*

We provide the formal proof in Appendix Sec. A.1. The accumulative property is essential for efficiency. If the accumulative condition is satisfied, only the aggregation result $x^k = g(\Delta x^0, \ldots, \Delta x^k)$ needs to be maintained and is updated by accumulating a new Δx^{k+1}, i.e., $x^{k+1} = g(x^k, \Delta x^{k+1})$. Otherwise, the large set ΔX^k needs to be maintained for every recursion k. The accumulative property will be used in our system design to save a lot of maintaining cost.

For example, the SSSP computation (Example 1) can be executed as an accumulated recursive program. Its non-aggregate operation $f()$ on node i outputs not only the tuples set $\{\langle j, td_j^{k+1} \rangle\}$ for its outgoing neighbors j but also its previous recursion's aggregation result $\langle i, d_i^k \rangle$. That is, $\{\langle i, td_i^{k+1} \rangle\}$ and $\langle i, d_i^k \rangle$ are contained in X^{k+1}, while $\langle i, d_i^k \rangle$ is already contained in X^k. Hence, $X^k \subseteq X^{k+1}$ and the monotonic condition is satisfied. In addition, the aggregate operation $g()$ which is MIN has the accumulative and commutative conditions.

3.3 Conditions for Asynchronous Aggregation

As discussed, asynchronous aggregation is possible in accumulated recursive programs. However, not all accumulated recursive programs with asynchronous aggregation will return the same result as that with synchronous aggregation. We demonstrate the sufficient conditions for asynchronous aggregation in the following theorem.

THEOREM 3.4. *(**Asynchronizability**) With asynchronous aggregation, an accumulated recursive program will yield to the same result as with synchronous aggregation, as long as the following order independent condition is satisfied.*

- ***order independent:*** *$g \circ f \circ g(X) = g \circ f(X)$;*

By asynchronous aggregation, the partial aggregation result is immediately used by the next $f()$ operation. The order independent property implies that no matter the $f()$ operation is first applied or the $g()$ operation is first applied, the effect is the same. As long as the same number of $f()$ operations are applied on all data, the eventual aggregation result will be the same. The detailed proof is provided in Appendix Sec. A.2.

For the SSSP example, the $f()$ operation expands the BFS searching scope to one-hop-away nodes, and the $g()$ operation picks the minimal distance resulted from the shortest path. The shortest distances are the same no matter making expansion first $g \circ f(X)$ or making aggregation first $f \circ g(X)$, i.e., $min_j(d_j + w) = min_j(d_j) + w$. There are a broad class of computations that satisfy these conditions and can be executed asynchronously (13 programs in Appendix Sec. B).

3.4 Conversion Possibilities

Some non-monotonic computations cannot be written as accumulated recursive program and are not originally qualified for asynchronous aggregation. For example, the PageRank computation cannot be executed as an accumulated recursive program since neither the monotonic condition nor the accumulative condition is satisfied. After applying $f \circ g$ operations, the recursion result X^k is replaced by X^{k+1} but not contained in X^{k+1}. Further, the aggregate operation $g(\{x_i\}) = \sum_i x_i + 0.15$ does not have the accumulative property due to the additional constant 0.15.

Fortunately, these computations can be converted to be qualified for asynchronous aggregation. We provide the convertibility conditions as follows.

THEOREM 3.5. *(**Convertibility**) A recursive program can be converted to an accumulated recursive program for asynchronous aggregation, as long as the recursion performs infinite times and an aggregate operation $g'()$ with the **accumulative** and **commutative** properties can be found with the following conditions:*

- ***convertible:*** *$g(X) = g'(X \cup y)$;*
- ***eliminable:*** *$lim_{n\to\infty}(g' \circ f)^n(x) = \mathbf{0}$,*

*where the $g'()$ and $f()$ operations are **order independent** $g' \circ f \circ g'(X) = g' \circ f(X)$, y is a constant value, and $\mathbf{0}$ is the identity element of the $g'()$ operation. The $g'()$ operation is the aggregate operation in the new accumulated recursive program.*

The formal proof can be found in Appendix Sec. A.3. We present the intuition as follows. We aim to find a new aggregate operation $g'()$ that is with an additional constant input value y but always returns the same result as $g()$, i.e., $g(X) = g'(X \cup y)$. The constant input implies the set containment relation between result and input, which makes it satisfy the monotonic condition that is required for accumulated recursive program. In addition, the eliminable property guarantees the accumulated recursive program to converge, such that the result is independent of X. Note that, it is not realistic to perform recursion infinite times. In practice, we will stop the recursion as long as $(g' \circ f)^n(x)$ is "small" enough.

For example, PageRank is such an iterative algorithm that can be converted to the accumulated recursive form [55]. In original PageRank, $g(\{x_j\}) = \sum_j x_j + 0.15$. The $g()$ function can be rewritten as $g(X) = g'(X \cup y)$ where the $g'()$ function can be "SUM", $X = \{x_j\}$ is the input values set, and the constant value y can be 0.15. Apparently, $g'()$ (i.e., "SUM") has the accumulative and commutative property. In addition, the $f()$ operation in PageRank contains a constant damping factor 0.85, which leads to $(g' \circ f)(x) < x$ and $lim_{n\to\infty}(g' \circ f)^n(x) = 0$. Furthermore, the $f()$ operation and $g'()$ operation in PageRank is order independent, which makes it suitable for asynchronous aggregation. The resulted ranking scores are the same no matter making decay first $g \circ f(X)$ or making summation first $f \circ g(X)$, i.e., $\sum_j(0.85 * \frac{r_j}{d}) = 0.85 * \frac{\sum_j(r_j)}{d}$. There exist other convertable algorithms, such as Program 13 Jacobi Method in Appendix Sec. B.

3.5 Aggregations Scheduling

Due to the power-law property of real world data, the computations that rely on different data inputs also exhibit uneven contributions to the final result. Asynchronous aggregation offers us the scheduling flexibility of these computations but has the side effect of nondeterministic performance. Therefore, a scheduling policy is crucial to guarantee better performance.

In accumulated recursive program, the aggregate operation induces a partial order defined as follows.

Definition 3.6. (**Partial Order**) Let X and Y denote two subsets of a set S. In accumulated recursive program, the aggregate operation $g()$ induces a partial order $\preceq$ over S, such that $g(X) \preceq g(Y)$ if and only if $X \subseteq Y$.

For example, MIN and MAX operations induce partial orders "$\leq$" and "$\geq$" respectively. SUM and COUNT operations both induce partial order "$\geq$".

Let $g(X)$ be the accumulated result at some point. By asynchronous aggregation, $f \circ g$ operations are further applied on a subset $Y \subseteq X$. Due to the accumulative property, the accumulated result is updated to be $g(X \cup f \circ g(Y))$. Since $X \subseteq X \cup f \circ g(Y)$, we have $g(X) \preceq g(X \cup f \circ$

$g(Y)$) according to Definition 3.6. This implies that the accumulated result $g(X)$ is monotonically increasing with respect to the partial order $\preceq$. The accumulated result $g(X)$ will not stop increasing until a convergence point $x^* = g(X^*)$ is reached. We have the following theorem to guide the scheduling of aggregate operations.

Theorem 3.7. ***(Aggregations Scheduling)*** *In accumulated recursive program where $g()$ defines a partial order $\preceq$, let $g(X)$ be the accumulated result during the computation, $\{Y_1, \dots, Y_n\}$ be all available subsets of X that can perform asynchronous aggregation. By scheduling the $f \circ g$ operations on the subset Y_i such that $g(Y_j) \preceq g(Y_i)$ for any Y_j, the accumulated result $g(X \cup f \circ g(Y_i))$ is the closest to $g(X^*)$.*

Proof. Due to the monotonic property, f is monotone under g, then $f \circ g$ is also monotone under g. If $g(Y_j) \preceq g(Y_i)$ for any j, we have $g(f \circ g(Y_j)) \preceq g(f \circ g(Y_i))$. Further, we have $g(X \cup f \circ g(Y_j)) \preceq g(X \cup f \circ g(Y_i))$. Since the accumulated result is monotonically increasing w.r.t. $\preceq$, scheduling the $f \circ g$ operations on the subset Y_i will result in the biggest progress approaching to the convergence result $g(X^*)$. □

Take graph computation as an example, Y_i corresponds to the set of state values of node i's neighbors. In SSSP, the node i with the shortest distance (i.e., $g(Y_i)$ is the smallest where Y_i is the set of shortest distances of its neighbors) is always scheduled first, which is exactly the same as Dijkstra's algorithm. In PageRank, the node i with the largest ranking score (i.e., $g(Y_i)$ is the largest where Y_i is the set of ranking score increments of its neighbors) is always scheduled first, which was shown much better performance [54].

3.6 Automatic Asynchronization

Though the conditions for asynchronous aggregation have been identified, it is still hard for a non-expert programmer to verify these conditions manually. It is even harder to find an aggregate function $g'()$ in order to convert a normal recursive program to an accumulated recursive program. To alleviate the burden of programmers, an automatic asynchronization scheme is desired. In this subsection, we discuss how to automatically verify these conditions given the $g()$ and $f()$ operations[2].

The condition verification problem can be thought of as a form of the constraint satisfaction problem, which can be analyzed with the help of *satisfiability modulo theories* (SMT) [11]. The satisfiability of these conditions can then be checked by an SMT solver, such as Z3 [17]. Next, we show how to reduce these condition verification problems into SMT satisfiability solving problems.

First, the $f()$ and $g()$ functions and the conditions have to be translated into a formula for being verified by an SMT solver. Given a function $F = f(x_1)$ or $F = g(x_1, \dots, x_n)$, we are interested in a formula for F of the form $\phi_F^{o_1,\dots,o_m}$, where $o_1, \dots, o_m$ are output variables [31]. Intuitively, the formula ϕ is "correct" if an assignment to $\{x_1, \dots, x_n, o_1, \dots, o_m\}$ makes ϕ be true if and only evaluating $F(x_1, \dots, x_n)$ returns $o_1, \dots, o_m$. Then the satisfiability of ϕ exactly implies that the function will compute the same output. Based on this formula, we present the formulas for different conditions in the following.

[2]The g and f operations in Datalog programs can be identified as will be shown in next section.

For the accumulative and commutative conditions in Theorem 3.3 and the order independent condition in Theorem 3.4, the condition verification problems can be reduced to SMT satisfiability problems via the following theorem.

Theorem 3.8. ***(Accumulative, Commutative, and Order Independent)*** *$\forall x_1, x_2, x_3$, the condition 1 or 2 or 3 is true if and only if $\phi_{F_l}^{o_1,\dots,o_m} \wedge \phi_{F_r}^{o'_1,\dots,o'_m} \wedge (\vee_{i=1}^m o_i \neq o'_i)$ is not satisfiable, where*

- *condition 1: accumulative, $F_l = g(x_1, x_2, x_3)$ and $F_r = g(g(x_1, x_2), x_3)$;*
- *condition 2: commutative, $F_l = g(x_1, x_2)$ and $F_r = g(x_2, x_1)$;*
- *condition 3: order independent, $F_l = g(f(g(x_1, x_2)))$ and $F_r = g(f(x_1), f(x_2))$.*

In the theorem, $\wedge$ denotes AND operator and $\vee$ denotes OR operator. Note that, SMT solver cannot answer "whether a formula F is always true?" but only answers "whether a formula F is satisfiable?". A formula F is *satisfiable* if there is some assignment of appropriate values to its uninterpreted function and constant symbols under which F evaluates to true. Thus, to verify a condition (that should be always true), we convert the "F is always true" problem to the "not F is not satisfiable" problem. If F is always true, then "not F" is always false, and then "not F" will not have any satisfying assignment

Theorem 3.9. ***(Eliminable)*** *$\forall x$, the eliminable condition is true if $\phi_F^{o_1,\dots,o_m} \wedge (\vee_{i=1}^m o_i \preceq x)$ is not satisfiable, where $F = g'(f(x))$ and $\preceq$ is the partial order induced by $g'()$.*

Similarly, we use Theorem 3.9 to verify the eliminable condition in Theorem 3.5. The intuition is that the contraction property $g'(f(x)) \preceq x$ implies the eliminable property, which can be used as a sufficient condition for verifying the eliminable condition.

Theorem 3.10. ***(Monotonic)*** *$\forall x_1, x_2$, the monotonic condition is true if and only if $\phi_F^{o_1,\dots,o_m} \wedge ((\vee_{i=1}^m x_1 \neq o_j) \vee (\vee_{i=1}^m x_2 \neq o_j))$ is not satisfiable, where $F = f(g(x_1, x_2))$.*

We use Theorem 3.10 to verify the monotonic condition in Theorem 3.3. The formula implies that, neither are x_1, x_2 contained in the result set nor is any of them contained in the result set.

Theorem 3.11. ***(Convertable)*** *$\forall x_1, x_2$, the convertable condition is true if $\phi_{F_l}^{o_1,\dots,o_m} \wedge \phi_{F_r}^{o'_1,\dots,o'_m} \wedge (\vee_{i=1}^m o_i \neq o'_i)$ is satisfiable, where $F_l = g(x_1, x_2)$, $F_r = g'(x_1, x_2, c)$, and c is a constant.*

Furthermore, SMT solver Z3 can be used to find a qualified aggregate function $g'()$ in Theorem 3.5. We declare a constant c and declare a function $g'()$ without defining

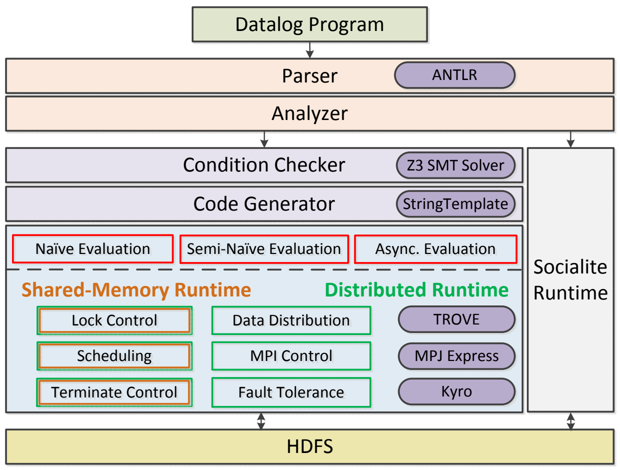

Figure 1: A3Log overview

the specific computation. Following Theorem 3.11, we use Z3 to check the satisfiability. If the formula is satisfiable, it will return the possible $g'()$ and c, otherwise it will return unknown or unsatisfiable. We can then convert the returned $g'()$ to a program to achieve automatic conversion.

4 A3LOG SYSTEM

In this section, we propose a Datalog System A3Log to support asynchronous aggregation, which is built by modifying Socialite [30, 39]. Datalog is an excellent candidate language for expressing computation logic because of its high-level *declarative* semantics and support for recursion. In particular, Datalog program only defines how the output depends on the inputs but not how the outputs are produced. It operates on a relation at a time, exposing plenty of opportunities for parallelization and asynchronization by virtue of the size of the data sets, so that we have the flexibility to design specific runtime engines to support *automatic* parallelization and *automatic* asynchronization. In addition, the program size of Datalog is much smaller than that of the imperative languages such as Java, C++. For these reasons, we choose Datalog as the high-level language.

A3Log is implemented using Java and contains several components as shown in Fig. 1. The **Parser** parses user's Datalog program into an abstract syntax tree (AST) by using ANTLR [5]. The **Analyzer** traverses the AST to performs syntactic and semantic analysis, identifies the recursive rule, and analyzes the aggregate operation g and non-aggregate operation f. If the program is a recursive program, it will be further processed by our system, otherwise it will be executed by the Socialite runtime engine [30, 39]. Given the f and g operations retrieved by the Analyzer, the **Condition Checker** relies on Z3 SMT Solver [17, 51] to verify the conditions and automatically chooses the appropriate evaluation technique. The **Code Generator** provides a series of code templates for generating the share-memory runtime code and distributed runtime code, where we use StringTemplate [45] to generate source code. Then, the recursive program will be executed by our execution engine (Sec. 4.4). A3Log provides both **Shared-Memory Runtime Engine** and **Distributed Runtime Engine**. The shared-memory runtime and distributed runtime share several functionalities, including lock control, scheduling, termination check. The distributed runtime additionally supports data distribution, MPI control, and fault tolerance. We use TROVE [47] for high performance container operations which are frequently used due to our key table structure, use MPJ Express [36] for MPI communication, and use Kryo [29] for efficient serialization and deserialization. As implemented in Socialite, A3Log also relies on HDFS to store data and to checkpoint intermediate computation state.

4.1 Datalog and Extension

A Datalog program is a set of rules. A *rule* **r** has the form $h \leftarrow b_1, \ldots, b_n$, where h is the *head* and $b_1, \ldots, b_n$ is the *body*. The comma separating literals in a body is a logical conjunction (AND). h and b_i can be with the form $p_i(t_1, \ldots, t_j)$ where p_i is a *predicate* and $t_1, \ldots, t_j$ are terms which can be *constants*, *variables* or *functions*. For example, the Datalog program for SSSP computation can be written as follows.

```
Program 1. Single Source Shortest Path
r1. sssp(X,d)← X=1,d=0.
r2. sssp(Y,min[d])← sssp(X,d1),edge(X,Y,d2),
                    d=d1+d2,sssp(Y,d).
```

In Program 1, rule `r1` initializes the predicate `sssp` by specifying the source node $X = 1$ and the shortest distance from source as $d = 0$. `r2` is a recursive rule since it has the `sssp` predicate in both its head and body. `r2` will recursively produce `sssp` fact by joining the old `sssp` and `edge`. If there is a path from source to X of length d_1 and an edge from X to Y of length d_2, there is a path from source to Y with length $d = d_1 + d_2$. If there is already a path to Y found before, it should be also considered. Hence, the shortest distance from source to Y is updated by the minimum of these possible distances, i.e., $\min[d]$. The recursion will terminate as soon as no shortest distance is updated.

Originally, the Datalog program terminates when no new fact can be found. However, some programs will never stop since it continuously produces "tiny" facts. For example, the PageRank computation will continuously update the PageRank scores even though the changes are less and less. In order to help users express more termination conditions, we allow users to specify the termination conditions using aggregations.

```
Program 2 PageRank
r1. degree(X,count[Y])← edge(X,Y).
r2. rank(X,r) ← node(X),r=1.
r3. rank(Y,sum[r]+0.15) ← rank(X,r1),edge(X,Y),
                          degree(X,d),
                          r=0.85·r1/d,
                          [sum[Δr] ≤ 0.001].
```

In Program 2, we show the Datalog program for PageRank. `r1` computes the node degrees based on the edge data. `r2` initializes the predicate `rank` by specifying all the nodes'

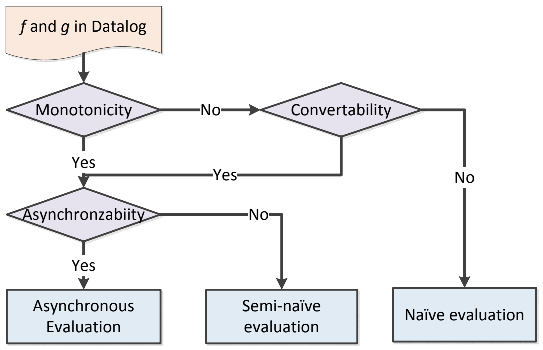

Figure 2: Condition checking flow chart

ranking scores as a constant 1. `r3` is a recursive rule that updates the predicate `rank`. We allow users to specify the terminate conditions in [. . .] by using different aggregate operations. In this program, the PageRank computation will terminate when the sum of ranking score differences of two continuous recursion results is less than or equal to 0.001.

Besides SSSP and PageRank, we list 11 more example Datalog programs that can be executed asynchronously in the Appendix Sec. B. These examples cover a wide range of applications, including graph analytics (Program 3, 4, 5, 12), data mining (Program 6, 7, 9, 10), machine learning (Program 8, 11), and HPC (Program 13).

4.2 Condition Checker

A3Log is able to automatically verify the conditions for asynchronous aggregation and accordingly selects the appropriate optimization technique according to the satisfiable conditions. The g operation is identified as the update operation in the recursive rule's head predicate, e.g., min[d] in SSSP and sum[r]+0.15 in PageRank where [d] and [r] are the inputs, respectively. The f operation is identified as the computation in the recursive rule's certain body predicate that updates the input variables of g operation, e.g., $d = d1 + d2$ in SSSP and $r = 0.85 \cdot r1/d$ in PageRank. The input variable of f operation is the variable that appears in the recursive predicate, e.g., $d1$ in `sssp(X,`$d1$`)` and $r1$ in `rank(X,`$r1$`)`.

There are two techniques for executing recursive Datalog programs. The first is ***Naive Evaluation***. For instance, in Program 1, the recursive rule `r2` will be repeatedly evaluated. The `edge` facts keep being joined with the `sssp` facts discovered so far in each iteration, until no new fact is produced. This approach will inefficiently re-produce known facts in every iteration. To address this inefficiency issue, the ***Semi-Naive Evaluation*** for recursive programs with aggregation was proposed [30, 49], which is efficient and produces no duplicates. However, the semi-naive evaluation requires the program to be set-containment monotonic in order to guarantee the correctness. This is the same as the monotonizability condition that we have defined in Theorem 3.3. Therefore, as long as the program satisfies monotonizability condition or can be converted to be monotonic according to Theorem 3.5, it can be executed with semi-naive evaluation. Furthermore,

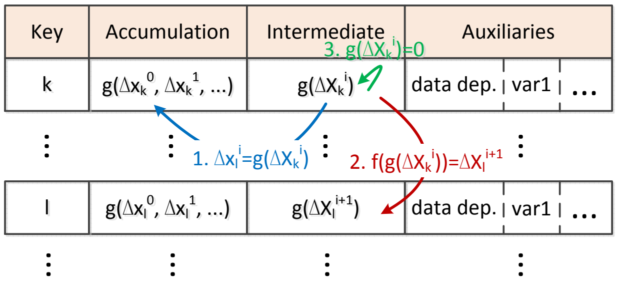

Figure 3: AsyncTable update

we propose ***asynchronous evaluation*** as a yet another optimization technique, which allows asynchronous aggregation. We provide the sufficient conditions for asynchronous aggregation in Theorem 3.4. The prerequisite for asynchronous aggregation is the monotonic condition. Asynchronous evaluation is possible only when the recursive program is monotonic and satisfies order independent condition.

Condition Checker first translates the f and g operations into Z3 [17] satisfiability formulas (see Sec. 3.6). By applying f and g to the built-in condition checking templates, the Z3 SMT olver will answer satisfiable, unsatisfiable, or unknown. Following the condition checking flow as shown in Fig. 2, our system can automatically select the appropriate evaluation technique.

4.3 Key Data Structure: AsyncTable

Before launching the recursive computation, the key table structure, AsyncTable, should be first prepared. AsyncTable maintains the computation states, which are initialized based on user's Datalog program and keeps being updated during the whole recursive computation process.

AsyncTable Design. As shown in Fig. 3, the AsyncTable contains several columns. The main key column and accumulation column store the result key-value pairs. The main key entries index the table. The accumulation column entry monotonically aggregates the intermediate results and maintains $g(\Delta x^0, \Delta x^1, \ldots)$. The accumulative property of g operation makes it possible to use a single column to maintain all the intermediate results. The intermediate column entry stores the temporary aggregation results $g(\Delta X^k)$ which will be used to generate future intermediate results by applying f operation. The auxiliary columns store the data which might be used by the f or g operation, e.g., outgoing neighbors set in SSSP and the node degree value in PageRank. The accumulation column entries and intermediate column entries are volatile, which maintain the computation states and keep being updated during the computation, while the other column entries are fixed after initialization. The AsyncTable is sharded for parallel processing. Each shard contains a number of rows and is assigned to a compute thread or placed on a distributed worker machine.

AsyncTable Initialization. The AsyncTable is initialized according to user's program. The recursive rule head contains the main key and accumulation column's information. For example in SSSP, the main key column entries are

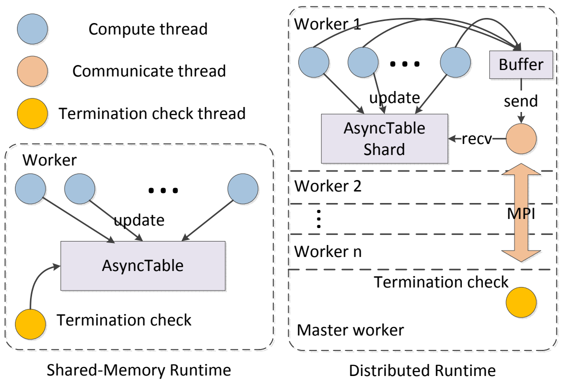

Figure 4: Execution Runtime Engine

initialized with the node ids. The accumulation column is initialized with the identity element **0** w.r.t g, e.g., MAX in SSSP. The intermediate column is initialized in terms of the non-recursive rule `r1`, i.e., 0 for the source node and MAX for other nodes. The auxiliary data dependency column is identified as the rule body predicate that describes the relationship between main keys, e.g., `edge(X,Y,d2)`. The auxiliary variable column entries can be identified as the joined results between rule body predicates, e.g., the degree information `d` in PageRank.

A few additional facts should be noticed: 1) It is possible that two or more items are identified as the main key (see Program 4, 5, and 12 in Appendix Sec. B); 2) The AsyncTable size can be dynamic rather than fixed due to the continuously inserted new rows (see Program 4, 5, 7, and 9); 3) The recursive program can be composed of more than one interdependent rules, which can be reduced to one rule (see Program 9 and 10).

4.4 Execution Runtime Engine

Overall Architecture. A3Log provides both shared-memory runtime engine and distributed runtime engine. Fig. 4 shows the overall structure. The shared-memory runtime has a simple design. A number of compute threads are created to update the AsyncTable in parallel, and a termination check thread is running separately to check the stop condition. The distributed runtime contains a number of workers and a master worker. Each worker creates a number of compute threads to update the local AsyncTable shard. A communicate thread tackles the remote update. We adopt a send buffer design to control the overhead of frequent communications. The master worker creates a termination check thread to globally check stop condition periodically.

Concurrency Control. The accumulation column entries and intermediate column entries keep being updated during computation. Specifically, an intermediate column entry is read to update the same row's accumulation column entry (by operation 1 in Fig. 3) and to update other rows' intermediate column entries (by operation 2 in Fig. 3), and is reset as **0** (by operation 3 in Fig. 3). The reset operation is essential to make result correct, without which the same intermediate result would be redundantly accumulated. These three operations have to be *atomic* according to the definition of accumulated recursive program (Definition 2). However, an intermediate column entry can be read and written concurrently by multiple threads or multiple workers, which leads to potential consistency problem. In our shared-memory runtime, we use *optimistic* lock to avoid read-write conflicts. In distributed runtime, the consistency problem will happen in the send buffer, which maintains the remote updates. The optimistic lock does not work well since the communicate thread is required to serialize the whole buffer for efficiency and the lock granularity becomes too coarse to result in too many aborts. We employ MVCC (multi-version concurrency control) to address this problem. Specifically, we adopt a double-version design of the send buffer, which serves the compute threads and communicate thread alternatively.

Scheduling. As discussed in Sec. 3.5, the clever scheduling of aggregate operations has the potential of accelerating the convergence of recursive computations. The scheduling should be performed by evaluating the intermediate aggregation result (intermediate column entries in AsyncTable). The top intermediate column entries (in the partial order defined by g) should be scheduled. Since the intermediate column entries keep being updated, the top entries should be evaluated frequently. For the sake of reducing scheduling cost, we utilize the sampling technique [54] to approximately obtain the top $N\%$ entries. The scheduling is even more costly in distributed environment. Rather than global scheduling, local scheduling in each worker is preferred.

Termination Check. The termination condition might rely on aggregation of either the accumulation column entries or the intermediate column entries. The aggregation is evaluated by the termination check thread without disturbing the compute threads. While in distributed runtime, each worker will report their local aggregation results to the master, where the global termination check thread determines whether to stop by evaluating the global aggregation result. Note that, there is no iteration's conception in asynchronous aggregation, so we have to check the termination condition every other period of time rather than every other iteration.

Fault Tolerance. For large scale operations that involve many machines for a substantial amount of time, it is also important to provide fault tolerance support. A3Log relies on Socialite's fault tolerance scheme [39]. The intermediate computation states are checkpointed occasionally on HDFS and restorable as needed. If one or more workers fail, the intermediate states are restored from the latest checkpoint and the evaluation is resumed from that point.

5 PERFORMANCE EVALUATION

We empirically evaluate A3Log on Amazon EC2. We will show the comparison results with other systems, with different algorithms, and with different datasets. More experimental results can be found in Appendix Sec. C, including scaling performance and effectiveness of aggregations.

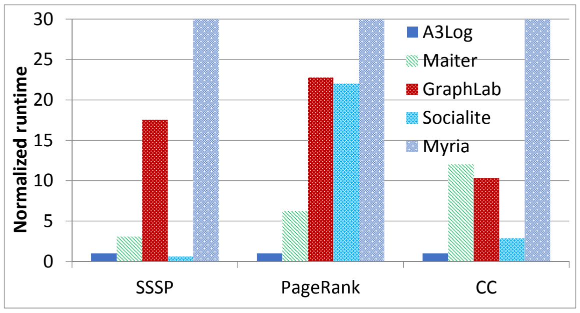

Figure 5: Performance comparison with other systems (many-core environment)

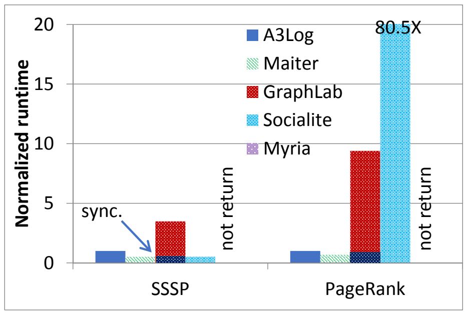

Figure 6: Performance comparison with other systems (distributed environment)

5.1 Comparison to Other Systems

Competitors. A3Log is compared with four state-of-the-art parallel/distributed frameworks. **SociaLite** [30, 39] is a Datalog implementation for social network analysis. **Myria** [22, 49] supports Datalog asynchronous evaluation. Both SociaLite and Myria support monotonic aggregation inside recursion. **GraphLab** [33] is a graph-based parallel/distributed engine supporting asynchronous iteration. **Maiter** [55] supports delta-based accumulative iterative computation (similar to monotonic aggregation) which can be executed asynchronously. All these systems are configured with their default parameters. Myria, GraphLab, and Maiter are configured to run with asynchronous model unless particularly mentioned. The runtime results are the execution time excluding data loading time, and each is a mean of two runs.

Algorithms and Datasets. We compare A3Log with other systems in the context of three algorithms, including SSSP (Program 1), PageRank (Program 2), and Connected Components (CC, Program 3). These three algorithms are all supported by these systems. For SSSP and CC, the computations terminate as soon as no new update is found. For PageRank in A3Log and Maiter, which has no notion of iterations, the computation terminates when the sum of difference to the theoretical convergence point [54] is less than 10^{-4} (see Appendix Sec. C.2), based on which we know the number of iterations that is required to reach the same point (42 iterations) [54]. Socialite and Myria (Myria does not support asynchronous PageRank) are set to run 42 iterations. GraphLab terminates after the PageRank value of every vertex changes by less than a user-specified threshold ϵ between two consecutive executions of that vertex. These algorithms are performed on a large graph dataset ClueWeb09 [14]. In order to finish the experiments in a reasonable time, we truncate the original dataset into a smaller ClueWeb20M dataset with 20,000,000 nodes, 243,063,334 edges, and 3.8GB size. Since ClueWeb is an unweighted graph, we assign a random weight to each edge for SSSP computation.

In the shared-memory experiment, we run all the systems on an r3.8xlarge EC2 instance with 32 vCPUs and 244GB memory. We configured all these systems with 32 threads. The normalized runtime results. In general, A3Log outperforms all the competitors. For the PageRank computation, A3Log achieves 22.8X speedup over GraphLab, 22X speedup over Socialite, 6X speedup over Maiter, and much more speedup over Myria[3] are shown in Fig. 5. There is an exception for SSSP computation. Socialite is 1.6X faster than A3Log. This may be due to their *prioritization* optimization, which is similar to our scheduling technique that leads to Dijkstra algorithm.

In the distributed experiment, we deploy all the systems on an EC2 cluster with 64 c4.2xlarge EC2 instances, each with 8 vCPUs and 15GB memory. The network bandwidth between instance workers is 10Gb/s[4]. We configured all these systems with 8 threads for each worker. The normalized runtime results of SSSP and PageRank are shown in Fig. 6. A3Log outperforms GraphLab and Socialite on PageRank computation. The synchronous version of GraphLab performs much better than asynchronous GraphLab, which is due to its costly distributed locking [23, 33]. Socialite performs SSSP faster. The distributed Myria runs unexpected long without returning results, say 9 hours for PageRank and did not return. The performance of A3Log and Maiter is comparable on these two applications. The reason why A3Log does not show significant better performance in distributed environment is because of the expensive communication overhead. The communication module relies on a pure Java implementation of MPI, MPJ express [36], which shows about 10 times lower performance than native C implementation of Open MPI [13]. We adopt MPJ for its good compatibility but at the expense of performance. We are implementing an alternative distributed runtime based on Open MPI [38].

5.2 Performance Gain Analysis

In order to analyze the factors for performance improvement and eliminate the interference from system implementation factors, we also implement a synchronous version of A3Log, which uses synchronous semi-naive evaluation (i.e., synchronous accumulated recursive programs). In addition, we turn off the scheduling then it shows the performance ahieved by pure asynchronous aggregation.

Algorithms and Datasets. We test more algorithms to see the effect variations on different workloads, including Least

[3]The runtime results of Myria may be wrong because they are all unexpected long, say 135 times longer for PageRank and 321 times longer for SSSP than A3Log. We are contacting with Myria authors to fix it but cannot find the problem before the submission.
[4]The network specification does not clearly describe the bandwidth but is only classified as *High*, which is estimated as 10Gb/s.

Table 1: Performance when varying workloads (shared-memory runtime, d is diameter, e is powerlaw exponent)

Dataset	Nodes	Edges	Graph type	d	e	SSSP			PageRank		
						Sync.	Async.	Speedup	Sync.	Async.	Speedup
Actor	382,219	33,115,812	collaborate	13	2.131	34.62s	1.54s	22.44X	63.62s	7.03s	9.05X
Amazon	403,394	3,387,388	co-purchase	25	3.151	79.18s	1.55s	51.05X	53.77s	1.02s	52.9X
arXiv	28,093	4,596,803	co-author	9	1.731	19.7s	1.51s	13.03X	53.2s	1.01s	52.7X
DBLP	1,314,050	18,986,618	co-author	24	3.221	55.38s	3.18s	17.41X	89.76s	5.06s	17.75X
Flicker	2,302,925	33,140,017	social	23	1.711	38.3s	3.95s	9.71X	84.24s	24.2s	3.48X
Livejournal	5,204,176	49,174,464	social	23	1.537	57.26s	9.41s	6.08X	111.09s	49.43s	**2.25X**
Orkut	3,072,441	117,184,899	social	10	1.272	58.99s	14.07s	**4.19X**	187.74s	16.18s	11.6X
Patent-US	3,774,768	16,518,947	citation	26	4.001	66.83s	3.21s	20.8X	68.06s	6.11s	11.15X
Prosper	89,269	3,394,979	loan	8	2.191	37.75s	1.51s	24.93X	52.61s	1.02s	51.58X
RoadCA	1,965,206	2,766,607	road net	865	8.991	140.43s	1.55s	90.31X	57.06s	1.03s	**55.59X**
RoadTX	1,379,917	1,921,660	road net	1064	8.901	137.2s	1.55s	88.4X	54.49s	1.02s	53.63X
Skitter	1,696,415	11,095,298	Internet	31	2.291	31.09s	1.55s	20.01X	59.51s	3.04s	19.56X
soc-LiveJ	4,847,571	68,475,391	social	20	2.651	67.2s	11.2s	6X	113.51s	39.2s	2.9X
soc-Pokec	1,632,803	30,622,564	social	14	3.081	53.85s	4.81s	11.19X	76.83s	15.1s	5.09X
TREC	1,601,787	8,063,026	web	112	2.231	130.84s	1.6s	81.62X	55.29s	2.03s	27.23X
web-BerkStan	685,230	7,600,595	web	208	2.601	692.08s	3.1s	**222.82X**	54.13s	2.02s	26.76X
web-Google	875,713	5,105,039	web	24	2.731	70.8s	1.59s	44.64X	54.24s	2.03s	26.67X
Wiki-Talk	2,987,535	24,981,163	communicate	9	1.811	23.81s	3.1s	7.68X	83.57s	28.27s	2.96X
Youtube-u	3,223,589	9,375,374	social	31	2.211	30.6s	2.43s	12.6X	65.91s	5.08s	12.96X
Zhishi-Baidu	2,141,300	17,794,839	web	20	2.291	49.19s	3.29s	14.96X	66.72s	8.09s	8.25X

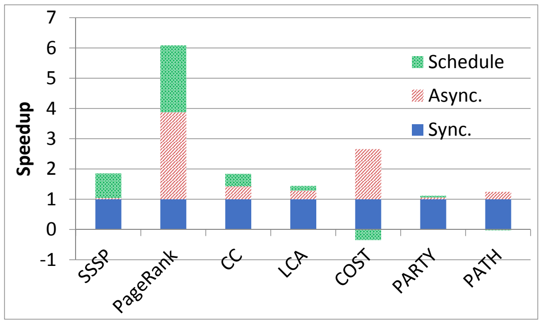

Figure 7: Performance gain analysis

Common Ancestor (LCA, Program 6), "What is the cost of each part?" (COST, Program 7), "Who will come to the party?" (PARTY, Program 9), and "Computing Paths in a DAG" (PATH, Program 4). More details of these algorithms can be found in Appendix Sec. B. For LCA, we use a citation network Patent-US [27]. For COST, we synthetically generate a hierarchical (tree-like) dataset with 30,000,000 tree nodes. PARTY is a graph based algorithm, so we use the same ClueWeb20M dataset. For PATH, we synthetically generate a directed acyclic graph (DAG) dataset with 500 nodes and 35,952 edges. PATH is a computation intensive workload since it evaluates the paths between all pairs.

We run these algorithms on the an r3.8xlarge EC2 instance with 32 vCPUs and 244GB memory. All these experiments are run with 32 threads. Fig. 7 shows the results. The runtime by synchronous semi-naive evaluation is considered as the baseline. The speedups from asynchronous execution and prioritized scheduling exhibit variations for different workloads. Generally speaking, the graph based algorithms benefit from asynchronous aggregation and priority scheduling more. For SSSP and PageRank, great performance gains are achieved by priority scheduling. However, for COST, the priority scheduling brings negative effect. This is because that COST aims to compute all parts' cost in a hierarchical structure and the computations on these parts equally contribute to the output. Using priority scheduling will not bring any benefit but only incurs scheduling overhead. However, we still take advantage of asynchronous aggregation to avoid synchronizations and achieve better performance.

5.3 Comparison with Different Workloads

To see the performance when varying datasets, we choose 20 various graph datasets with various graph structures and various graph properties. All the datasets are downloaded from [27]. We use two typical graph algorithms SSSP and PageRank for evaluation. We run the shared-memory version of A3Log on a c4-2xlarge EC2 instance with 8 vCPU and 60GB memory. A3Log is configured with 8 threads. We compare the algorithm runtime of synchronous execution and asynchronous execution on these graphs.

Table 1 shows the graph datasets and the runtime results. The asynchronous execution exhibits 4.19X-222.82X speedup over synchronous execution on SSSP computation, and 2.25X-55.59X speedup over synchronous execution on PageRank computation. Generally speaking, asynchronous SSSP achieves higher speedup on large diameter graphs, and asynchronous PageRank computation achieves higher speedup on the graphs with larger powerlaw exponent. Of course, the performance speedup also relates to the graph structures and graph types. Note that, in asynchronous execution, the termination check is performed periodically (every 1 second in this experiment). If the runtime results of asynchronous executions shows 1.x second, they may converge less than 1 second. Thus, the performance of asynchronous execution is expected to be even higher.

6 RELATED WORKS

Coordination Avoidance. Minimizing coordination, or blocking communication between concurrently executing operations, is key to maximizing scalability, availability, and high performance in database systems. However, uninhibited coordination-free execution can compromise application correctness, or consistency. Coordination and consistency are the most critical issues for system performance and manageability at scale [8]. Hellerstein et al. have set up the foundation [24] and have put a lot of efforts to advance this field [1, 10]. Dedalus [4] is proposed as a declarative foundation for the two signature features of distributed systems: mutable state, and asynchronous processing and communication. CALM (Consistent and Logical Monotonicity) principle [3] is described for reasoning about distributed system behaviour, which ensures eventual consistency by enforcing a *monotonic* logic. A declarative language called Bloom [16] that encourages CALM programming and is well-suited to the inherent characteristics of distribution. Edelweiss [15] is a sublanguage of Bloom that provides an Event Log Exchange (ELE) programming model, yet automatically reclaims space without programmer assistance, which can be used to elegantly implement asynchronous communications. Blaze [2] ensures consistent outcomes via a more efficient and manageable protocol of asynchronous point-to-point communication between producers and consumers. MacroBase [9] is a fast data system built to explore the fast data principles that suggests asynchronously prioritizing computation on inputs that most affect outputs.

Asynchronous Computation in Graph Analytics. Asynchronous computation has attracted much attention in the field of graph processing. GraphLab [33] aims to express asynchronous iterative algorithms with sparse computational dependencies while ensuring data consistency and achieving good parallel performance. Frog [40] is a lock-free semi-asynchronous parallel graph processing framework with a graph coloring model. Grace [48] is a single-machine parallel graph processing platform that allows customization of vertex scheduling and message selection to support asynchronous computation. Giraph++ [46] not only allows asynchronous computation while keeping the vertex-centric model but also is able to handle mutation of graphs. GiraphUC [23] relies on barrierless asynchronous parallel (BAP), which reduces both message staleness and global synchronization. Maiter [55] proposes delta-based asynchronous iterative computation model (DAIC) and supports distributed asynchronous graph processing. GunRock [50] supports fast asynchronous graph computation in GPUs. Unfortunately, the above graph systems do not support automatic asynchronization.

GRAPE [19] differs from prior systems in its ability to automatically parallelize existing sequential graph algorithms as a whole. Sequential graph algorithms can be "plugged into" GRAPE with minor changes, and get parallelized. As long as the sequential algorithms are correct, the GRAPE parallelization guarantees to terminate with correct answers under a monotonic condition. However, GRAPE cannot automatically asynchronize a sequential graph algorithm.

Asynchronous Computation in Machine Learning. In machine learning, some algorithms with non-serializable lock-free implementations offer significant speedups. Gonzalez et al. [21] examine the growing gap between efficient machine learning algorithms exploiting asynchrony and fine-grained communication, and commodity distributed dataflow systems (Hadoop and Spark) that are optimized for coarse-grained models. Google DeepMind group recently proposes asynchronous methods for deep reinforcement learning [34], which surpasses the current state-of-the-art on the Atari domain while training for half the time on a single multi-core CPU instead of a GPU. DimmWitted [53] exposes a range of options for asynchronous data sharing, which outperforms general cluster compute frameworks by orders of magnitude in the context of popular models, including SVMs, logistic regression, Gibbs sampling, and neural networks. HOGWILD! [37] is a implementation of stochastic gradient descent (SGD), which places a single copy of the model in the memory of a multi-core server and runs multiple worker processes that simultaneously run gradient steps in parallel. An asynchronous parallel stochastic coordinate descent algorithm is proposed in [32], which shows significant speedup in multi-core environment. A burgeoning cottage industry of machine learning researchers has begun to extend asynchronous execution strategies to an increasing number of optimization tasks.

Datalog Systems. Besides SociaLite [30, 39] and Myria [22, 49], there exist other Datalog systems. DeALS [42, 43] is a deductive database systems relying on Datalog language, supporting optimized execution over diverse platforms including sequential implementations, multi-core machines, and clusters. BigDatalog [41] is built on top of Spark [52] and provides declarative semantics of monotonic aggregate programs. Datalography [35] incorporates optimization techniques for efficient distributed evaluation of Datalog queries on Giraph [6]. LogicBlox [7] is a commercial database system based on LogiQL.

7 CONCLUSIONS

We have presented A3, Automatic Asynchronous Aggregation. We clearly define the conditions for asynchronous aggregation and propose to prioritize the aggregate computations that most affect result. We also propose a technique to allow automatic asynchronization. We further design and implement a Datalog system, A3Log, to support automatic asynchronous computation of the user's program. A3Log provides both shared-memory runtime and distributed runtime. Our results show that A3Log outperforms the state-of-the-art systems for various applications. Asynchronous execution is shown to achieve up to two-orders-of-magnitude speedup over synchronous execution.

REFERENCES

[1] Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. 2013. Consistency Without Borders. In *SOCC '13*. 23:1–23:10.

[2] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. 2014. Blazes: Coordination analysis for distributed programs. In *ICDE '14*. 52–63.

[3] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR '11*.

[4] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog '10*. 262–281.

[5] ANTLR 2017. ANother Tool for Language Recognition. (2017). http://www.antlr.org/.

[6] Apache Giraph 2017. Iterative graph processing system. (2017). http://giraph.apache.org.

[7] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD '15*. 1371–1382.

[8] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196.

[9] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. 2017. MacroBase: Prioritizing Attention in Fast Data. In *SIGMOD '17*. 541–556.

[10] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2014. Quantifying Eventual Consistency with PBS. *Commun. ACM* 57, 8 (Aug. 2014), 93–102.

[11] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. *Satisfiability modulo theories* (1 ed.). Frontiers in Artificial Intelligence and Applications, Vol. 185. 825–885.

[12] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (March 2017), 48–54.

[13] Alexey Cheptsov. 2012. Enabling High Performance Computing for Semantic Web Applications by Means of Open MPI Java Bindings. In *SEMAPRO '12*. 11–19.

[14] ClueWeb09 2017. The ClueWeb09 Dataset. (2017). https://lemurproject.org/clueweb09/.

[15] Neil Conway, Peter Alvaro, Emily Andrews, and Joseph M. Hellerstein. 2014. Edelweiss: Automatic Storage Reclamation for Distributed Programming. *Proc. VLDB Endow.* 7, 6 (Feb. 2014), 481–492.

[16] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *SoCC '12*. 1:1–1:14.

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS '08/ETAPS '08*. 337–340.

[18] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04*.

[19] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. 2017. Parallelizing Sequential Graph Computations. In *SIGMOD '17*. 495–510.

[20] Wolfgang Gatterbauer, Stephan Günnemann, Danai Koutra, and Christos Faloutsos. 2015. Linearized and Single-pass Belief Propagation. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 581–592.

[21] Joseph E. Gonzalez, Peter Bailis, Michael I. Jordan, Michael J. Franklin, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2015. Asynchronous Complex Analytics in a Distributed Dataflow Architecture. *CoRR* abs/1510.07092 (2015).

[22] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. 2014. Demonstration of the Myria Big Data Management Service. In *SIGMOD '14*. 881–884.

[23] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Systems. *Proc. VLDB Endow.* 8, 9 (May 2015), 950–961.

[24] Joseph M. Hellerstein. 2010. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Rec.* 39, 1 (Sept. 2010), 5–19.

[25] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *SIGMOD '97*. 171–182.

[26] Glen Jeh and Jennifer Widom. 2002. SimRank: A Measure of Structural-context Similarity. In *KDD '02*. 538–543.

[27] KONECT 2017. The Koblenz Network Collection. (2017). http://konect.uni-koblenz.de/.

[28] F. R. Kschischang, B. J. Frey, and H. A. Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Trans. on Information Theory* 47, 2 (Feb 2001), 498–519.

[29] Kyro 2017. Kryo serializers for some jdk types. (2017). https://github.com/magro/kryo-serializers.

[30] Monica S. Lam, Stephen Guo, and Jiwon Seo. 2013. SociaLite: Datalog Extensions for Efficient Social Network Analysis. In *ICDE '13*. 278–289.

[31] Chang Liu, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Zhenyu Guo, and Thomas Moscibroda. 2014. Automating Distributed Partial Aggregation. In *SOCC '14*. 1:1–1:12.

[32] Ji Liu, Stephen J. Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. 2015. An Asynchronous Parallel Stochastic Coordinate Descent Algorithm. *J. Mach. Learn. Res.* 16, 1 (Jan. 2015), 285–322.

[33] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727.

[34] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *ICML '16*. 1928–1937.

[35] W. E. Moustafa, V. Papavasileiou, K. Yocum, and A. Deutsch. 2016. Datalography: Scaling datalog graph analytics on graph processing systems. In *IEEE BigData '16*.

[36] MPJ Express 2017. An open source Java message passing library. (2017). http://mpj-express.org/.

[37] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS '11*. 693–701.

[38] Open MPI 2017. Open Source High Performance Computing. (2017). https://www.open-mpi.org/.

[39] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1906–1917.

[40] Xuanhua Shi, Junling Liang, Sheng Di, Bingsheng He, Hai Jin, Lu Lu, Zhixiang Wang, Xuan Luo, and Jianlong Zhong. 2015. Optimization of Asynchronous Graph Processing on GPU with Hybrid Coloring Model. In *PPoPP '15*. 271–272.

[41] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *SIGMOD '16*. 1135–1149.

[42] A. Shkapsky, M. Yang, and C. Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE '15*. 867–878.

[43] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. 2013. Graph Queries in a Next-generation Datalog System. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1258–1261.

[44] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2015. Sequential Random Permutation, List Contraction and Tree Contraction Are Highly Parallel. In *SODA '15*. 431–448.

[45] StringTemplate 2017. Java template engine for generating source code. (2017). http://www.stringtemplate.org/.

[46] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.* 7, 3 (Nov. 2013), 193–204.

[47] TROVE 2017. High performance collections for Java. (2017). http://trove.starlight-systems.com/.

[48] Guozhang Wang, Wenlei Xie, Alan Demers, and Johannes Gehrke. 2013. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR '13*.

[49] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and Fault-tolerant Recursive Datalog Evaluation in Shared-nothing Engines. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1542–1553.

[50] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *PPoPP*

'16. 11:1–11:12.
[51] Z3 2017. SMT Solver. (2017). https://github.com/Z3Prover/z3.
[52] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud '10*.
[53] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-memory Statistical Analytics. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1283–1294.
[54] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2011. PrIter: A Distributed Framework for Prioritized Iterative Computations. In *SOCC '11*. 13:1–13:14.
[55] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2014. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *IEEE Trans. on Parallel and Distributed Systems* 25, 8 (Aug. 2014), 2091–2100.

A PROOFS

A.1 Proof of Theorem 3.3

Proof. Since $X^{k-1} \subseteq f \circ g(X^{k-1}) = X^k$, we have $X^k = X^{k-1} \cup \Delta X^k$ where $\Delta X^k = X^k \setminus X^{k-1}$. Let x^k denote the result of normal recursive program and $\bar{x}^k$ denote the result of accumulated recursive program after k recursions. Then we have

$$\begin{aligned} x^k = g(X^k) &= g(X^{k-1} \cup \Delta X^k) && (1)\\ &= \dots && (2)\\ &= g(\Delta X^0 \cup \dots \cup \Delta X^{k-1} \cup \Delta X^k) && (3)\\ &= g(g(\Delta X^0) \cup \dots \cup g(\Delta X^{k-1}) \cup g(\Delta X^k)) && (4)\\ &= g(\Delta x^0, \dots, \Delta x^{k-1}, \Delta x^k) = \bar{x}^k. && (5)\end{aligned}$$

Line (1)-(3) are true because of the monotonicity. Due to the accumulativity and commutativity, we have $g(X_1 \cup X_2) = g(g(X_1) \cup X_2) = g(X_2 \cup g(X_1)) = g(g(X_1) \cup g(X_2))$, from which Line (4) is true. Line (5) is true because of the definition of accumulated recursive program. □

A.2 Proof of Theorem 3.4

Proof. Let ΔX_1^k and ΔX_2^k denote two disjoint subsets of ΔX^k, where ΔX^k is the intermediate result of the kth recursion in accumulated recursive program. If the aggregation is performed across any two recursions (which can be easily extended to any multiple recursions), i.e., $g(\Delta X_1^k \cup \Delta X_1^l)$ where $k \neq l$, and the result is the same as that in synchronous aggregation, the theorem is proved.

$$\begin{aligned} &g(\Delta X^0 \cup \dots g(\Delta X_1^{k+1} \cup \Delta X_1^{l+1}) \cup \Delta X_2^{k+1} \cup \Delta X_2^{l+1} \dots \cup \Delta X^n) && (1)\\ =&g(\Delta X^0 \cup \dots g(\Delta X_1^{k+1} \cup \Delta X_1^{l+1}) \cup g(\Delta X_2^{k+1} \cup \Delta X_2^{l+1}) \dots \cup \Delta X^n) && (2)\\ =&g(\dots g(f \circ g(\Delta X_1^k) \cup f \circ g(\Delta X_1^l)) \cup g(f \circ g(\Delta X_2^k) \cup f \circ g(\Delta X_2^l)) \dots) && (3)\\ =&g(\dots g \circ f(\Delta X_1^k) \cup g \circ f(\Delta X_1^l) \cup g \circ f(\Delta X_2^k) \cup g \circ f(\Delta X_2^l) \dots) && (4)\\ =&g(\dots g \circ f(\Delta X_1^k \cup \Delta X_2^k) \cup g \circ f(\Delta X_1^l \cup \Delta X_2^l) \dots) && (5)\\ =&g(\dots f \circ g(\Delta X_1^k \cup \Delta X_2^k) \cup f \circ g(\Delta X_1^l \cup \Delta X_2^l) \dots) && (6)\\ =&g(\Delta X^0 \cup \dots \cup \Delta X^{k+1} \cup \Delta X^{l+1} \cup \dots \cup \Delta X^n) && (7)\end{aligned}$$

Line (2) is true because of the accumulative and commutative properties of g operation which is required by accumulated recursive program (i.e., $g(X \cup Y) = g(X \cup g(Y))$). Line (3) is true because of Equation (2). Line (4) is true because $g \circ f \circ g(X) = g \circ f(X)$. Line (5) is true because $g(g(X_1) \cup g(X_2)) = g(X_1 \cup X_2)$ (from accumulative and commutative properties) and the distributive property of f operation. Line (6) is true because $g \circ f(X) = g \circ f \circ g(X)$ and $g(g(X_1) \cup g(X_2)) = g(X_1 \cup X_2)$. Line (7) is the result of synchronous recursion. Therefore, by asynchronous aggregation, the accumulated recursive program will yield to the same result as synchronous aggregation. □

A.3 Proof of Theorem 3.5

Proof. Let $g(X^0) = x^0$, then we have

$$\begin{aligned} (g \circ f)^n(x^0) &= (g \circ f)^n \circ g(X^0) && (1)\\ =&g \circ f \circ g \circ \dots \circ g \circ f \circ g'(X^0 \cup y) && (2)\\ =&g \circ f \circ g \circ \dots \circ g'\Big((f \circ g')(X^0 \cup y) \cup y\Big) && (3)\\ =&g \circ f \circ g \circ \dots \circ g'\Big((f \circ g')(X^0) \cup f(y) \cup y\Big) && (4)\\ =&\dots && (5)\\ =&g'\Big((f \circ g')^n(X^0) \cup f^n(y) \cup \dots \cup f(y) \cup y\Big) && (6)\\ =&g'\Big(f^n(y), \dots, f(y), y\Big) && (7)\end{aligned}$$

Line (2) is true because $g(X) = g'(X \cup y)$. Line (3) is true because $g'(X_1 \cup X_2) = g'\Big(g'(X_1) \cup g'(X_2)\Big)$. Line (4) is true because the distributive property of $f()$. Since $\mathbf{0}$ is the identity of $g'()$ function, i.e., $g'(X \cup \mathbf{0}) = g'(X)$, Line (6) is true when the recursion performs infinite times, i.e., $n \to \infty$. Line (7) is in the result format of accumulated recursive program, which implies that the normal recursive program is converted to the accumulated recursive program. □

B DATALOG EXAMPLES

```
Program 3. Connected Conponents
r1. cc(X,X)← edge(X,_).
r2. cc(Y,min[v])← cc(X,v),edge(X,Y),
                  cc(Y,v).
```

Program 3 computes the connected components in a graph. Each vertex starts as its own connected component with its identifier. For all combinations of facts that satisfy the recursive bodies, the aggregate function `min` keeps and propagates only the current minimal component ID v for each vertex `Y`.

```
Program 4. Computing Paths in a DAG
r1. cpaths(X,Y,c) ← edge(X,Y), c=1.
r2. cpaths(X,Y,count[c]) ← cpaths(X,Z,c),
                           edges(Z,Y),
                           cpaths(X,Y,c).
```

Program 4 counts the paths between pairs of vertices in an acyclic graph. `r1` counts each edge as one path between its vertices. In `r2`, any `edge(Z,Y)` that extends from a computed path count `cpath(X,Z,`c`)` establishes c distinct paths from `X` to `Y` through `Z`.

```
Program 5. Max Probability Path
r1. reach(X,Y,P) ← net(X,Y,P).
r2. reach(X,Y,max[P]) ← reach(X,Z,P1),
                        reach(Z,Y,P2),
                        P = P1 * P2,
                        reach(X,Y,P).
```

Program 5 [42] computes the max probability path between two nodes in a network. The `net` predicate denotes the probability `P` of reaching `Y` from `X`.

```
Program 6. Least Common Ancestor

r1. ancestor(Y,X,1) ← cite(Y,X), X<seed.
r2. ancestor(Z,X,min[d]) ← ancestor(Z,Y,d'),
                           cite(Y,X),d = d' + 1,
                           ancestor(Z,X,d).
r3. LCA(p1,p2,min[max(  ← ancestor(p1,X,d1),
      d1,d2)],year,X)     ancestor(p2,X,d2),
                          Paper(X,year),
                          p1<p2.
```

Program 6 [49] computes the least common ancestor (LCA) for pairs of publications in a citations graph. An ancestor a of a paper `Z` is any paper that is transitively cited by `Z`, and the LCA a of two papers `X` and `Y` is the least ancestor of both `X` and `Y`.

```
Program 7. What is the cost of each part
r1. cost(Part,C) ← basic(Part,cost).
r2. cost(Part,sum[C]) ← assb(Part,Sub,n),
                        cost(Sub,c),
                        C = c * n,
                        cost(Part,C).
```

Program 7 [42] is for computing the cost of a part from the cost of its subparts. The `assb` predicate denotes each parts required subparts and number required, and basic denotes the parts cost.

```
Program 8. Viterbi Algorithm
r1. calcV(0,X,max(L)) ← s(0,EX),p(X,EX,L1),
                        pi(X,L2),L = L1 * L2.
r2. calcV(T,Y,max[L]) ← s(T,EY),p(Y,EY,L1),
                        T1 = T − 1,t(X,Y,L2),
                        calcV(T1,X,L3),
                        L = L1 * L2 * L3.
```

Program 8 [42] is the Viterbi algorithm for hidden Markov models. `t` denotes the transition probability `L2` from state `X` to `Y`; `s` denotes the observed sequence of length `L+1`; `p` denotes the likelihood `L1` that state `X (Y)` emitted `EX (EY)`. `r1` finds the most likely initial observation for each `X`. `r2` finds the most likely transition for observation `T` for each `Y`.

```
Program 9. Who will come to the party?

r1. coming(X) ← sure(X).
r2. coming(X) ← cntComing(X,N), N ≥ 3.
r3. cntComing(Y,count[*]) ← friend(Y,X),
                            coming(X).
```

In this program, some people will come to the party for sure, whereas others only join when at least three of their friends are coming. Program 9 [42] describes the retrieving process. With `cntComing`, each person watches the number of their friends that are coming grow, and once that number reaches three, the person will then come to the party too.

```
Program 10. Galaxy Evolution

r1. galaxies(1,gid) ← galaxies_seed(gid).
r2. galaxies(t+1,gid2) ← galaxies(t,gid1),
                         edges(t,gid1,gid2,c),
                         c≥threshold.
r3. edges(t,gid1,gid2, ← galaxies(t,gid1),
      count[*])          particles(pid,gid1,t),
                         particles(pid,gid2,t+1).
```

Program 10 [49] computes the history of a set of galaxies in an astrophysical simulation. The history of a galaxy is the set of past galaxies that merged over time to form the galaxy of interest at present day. The predicate `Particles(pid,gid,t)` holds the simulation output as a set of particles, where `pid` is a unique particle identifier, and `gid` is the identifier of the galaxy that the particle belongs to at time `t`. The `gid` values are unique only within their timesteps `t`, but a particle retains the same `pid` throughout the simulation.

```
Program 11. Belief Propagation

r1. G(v,0) ← E(v,_,_).
r2. B(v,c,b) ← E(v,c,b).
r3. G(t,i) ← G(s,i-1),A(s,t,w),¬G(t,_).
r4. B(t,c2,sum[b']) ← G(t,i),A(s,t,w),
                      B(s,c1,b),
                      G(s,i-1),H(c1,c2,h),
                      b'=w*b*h,
                      [sum[Δb'] < T];
```

Belief Propagation [28] is a message-passing algorithm for performing inference on graphical models, such as Bayesian networks and Markov random fields. In Program 11, `B` is the to-be-returned beliefs, `G` maintains the geodesic numbers, `A` is an input weighted network with initial beliefs `E`, and `H` is the coupling scores. `r3` and `r4` should be repeated evaluated where the geodesic number `i` is increasing, and stops when no more facts inserted into `G`. Note that, the shown Datalog program describes a single-pass belief propagation process since it does not allow loopy propagation (by `¬G(t,_)` in `r3`) [20].

```
Program 12. SimRank

r1. D(X,count[*]) ← E(X,Y).
r2. S(X,Y,1) ← E(X,Y).
r3. S(X,Y,sum[s']) ← S(X',Y',s),E(X,X'),E(Y,Y'),
                     D(X,d1),D(Y,d2),X≠Y,
                     s'= C/(d1*d2) *s,
                     [sum[Δs'] < T].
```

SimRank algorithm [26] is an algorithm to evaluate the similarities between node pairs in a graph. In Program 12, `S` maintains the similarities between node pairs, `E` is the edge data, `D` maintains the out degree information. The algorithm terminates when the sum of differences between two recursions is small enough.

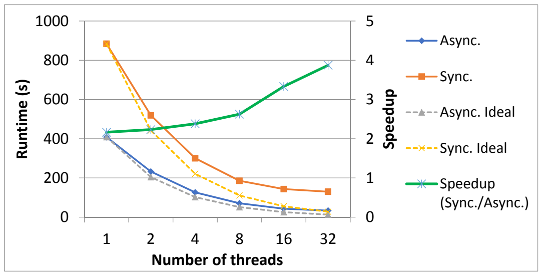

Figure 8: Scaling performance (many-core)

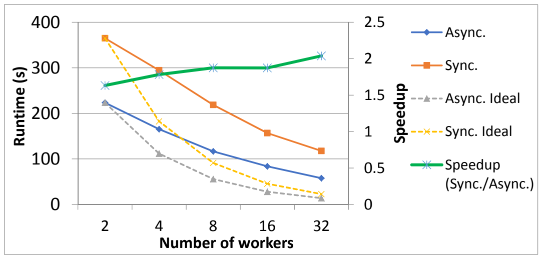

Figure 9: Scaling performance (distributed)

```
Program 13. Jacobi Method

r1. X(i,x) ← X(i,1).
r2. C(i,c) ← A(i,i,$a_{ii}$),B(i,$b_i$),
             c=$\frac{b_i}{a_{ii}}$.
r3. X(i,sum[x']+c) ← X(j,x),A(i,j,$a_{ij}$),
                     C(i,c),A(i,i,$a_{ii}$),i≠j,
                     x'=$-\frac{a_{ij}}{a_{ii}}$*x,
                     [sum[$\Delta x'$] $< T$].
```

In numerical linear algebra, the Jacobi method is a famous iterative algorithm for solving linear equations of the form $A \cdot X = b$, where A is a matrix with each entry a_{ij}, and b is a vector with each entry b_i. In Program 13, X is the solution vector, A is the matrix, B is the vector, C is a vector maintaining the entry-specific constant $\frac{b_i}{a_{ii}}$. A sufficient (but not necessary) condition for the method to converge is that the matrix A is strictly or irreducibly diagonally dominant, i.e., $|a_{ii} > \sum_{j \neq i} |a_{ij}||$, which implies the convertable condition, so Jacobi Method can be converted and executed asynchronously.

C MORE EXPERIMENTS

C.1 Scaling Performance

Due to coordination avoidance, asynchronous aggregation is expected to achieve better scaling performance. We evaluate the scaling performance of A3Log's share-memory runtime on an r3.8xlarge EC2 instance. We perform PageRank computation with the synchronous and asynchronous versions (without scheduling) of A3Log and vary the number of threads from 1 to 32. The runtime results are shown in Fig. 8. We also draw the ideal scaling performance curve based on the runtime result of 1 thread. The asynchronous A3Log's scaling performance curve is closer to its ideal curve, while the synchronous A3Log exhibits larger difference to its ideal curve when running on more threads. Asynchronous execution also shows higher speedup over synchronous execution when running on more threads.

We also evaluate the scaling performance of A3Log's distributed runtime on a cluster with a number of c4.large EC2 instances, each with 2 vCPUs and 3.75GB memory. We perform PageRank computation with the synchronous and asynchronous executions (without scheduling) and vary the number of workers from 1 to 32. The runtime results are shown in Fig. 9. We also draw the ideal scaling performance curve based on the runtime result of 2-workers. Similar to the results on shared-memory runtime, asynchronous execution also shows better scaling performance. Higher speedup over synchronous execution is achieved when running on larger size cluster.

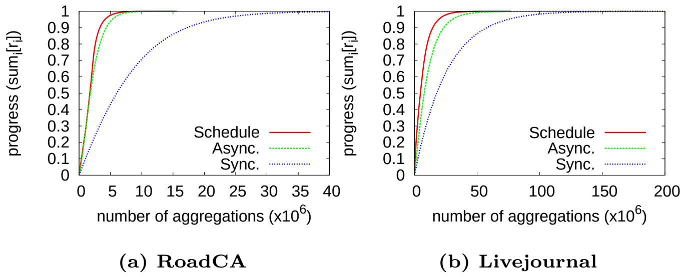

Figure 10: Effectiveness of aggregate operations to converging progress

C.2 Effectiveness of Aggregate Operations

The effectiveness of aggregate operations in asynchronous aggregation is an important advantage but tends to be underestimated. Synchronous aggregation requires to aggregate the information from previous iteration, while asynchronous aggregation accumulates the most up-to-date information. Thus, the aggregate operation is expected to be more effective. In this experiment, we evaluate the effectiveness of aggregations on accelerating the computation progress.

We run PageRank on two datasets, RoadCA and Livejournal, which result in the most speedup and the least speedup comparing to synchronous aggregation as shown in Table 1. We record the accumulated number of aggregate operations during the computation. In the accumulated version of PageRank (see Sec. 3.4), the summation of ranking scores $\sum_i r_i$ is approaching to 1 and will finally converge to 1. So we estimate the computation progress by evaluating $\sum_i r_i$ periodically. Fig. 10 shows the results. By using asynchronous aggregation, the converging progress is much faster than using synchronous execution. The scheduling of aggregate operations will further speedup the progress. The asynchronous aggregation with scheduling shows more effectiveness, so that each aggregate operation contributes more. It also shows more effectiveness on the RoadCA graph than on the Lievejournal graph. This is the reason why higher speedup is observed on the RoadCA graph.